University of Pittsburgh

# Algorithms and Data Structures 2
# CS 1501

Pitt SCI


Spring 2023

Sherif Khattab

ksm73@pitt.edu


(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

# Announcements

- Upcoming Deadlines
  - Homework 10: this Friday @ 11:59 pm
  - Lab 8: Tuesday 3/28 @ 11:59 pm
  - Assignment 3: Friday 3/31 @ 11:59 pm
    - Support video and slides on Canvas

# Previous lecture

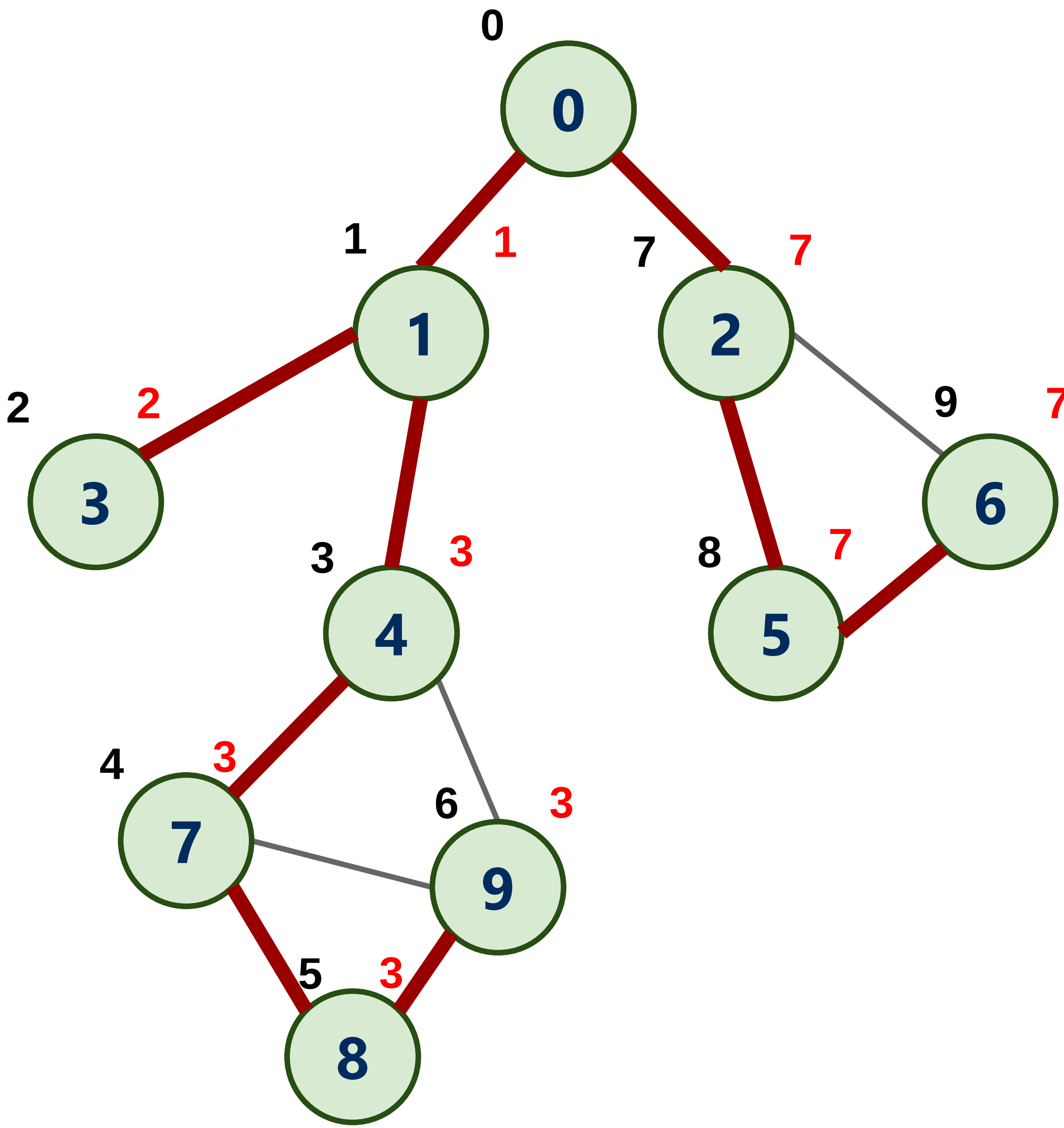
- ADT Graph
  - finding articulation points of a graph
  - Graph compression
  - Graphs with weighted edges
  - Minimum Spanning Tree (MST) problem

# This Lecture

- ADT Graph
    - Minimum Spanning Tree (MST) problem
        - Prim’s MST algorithm
        - Kruskal’s MST algorithm

# low(v)

- How do we find low(v)?
- low(v) = Min of:
  - num(v)
  - num(w) for all back edges (v, w)
  - low(w) of all children of *v*

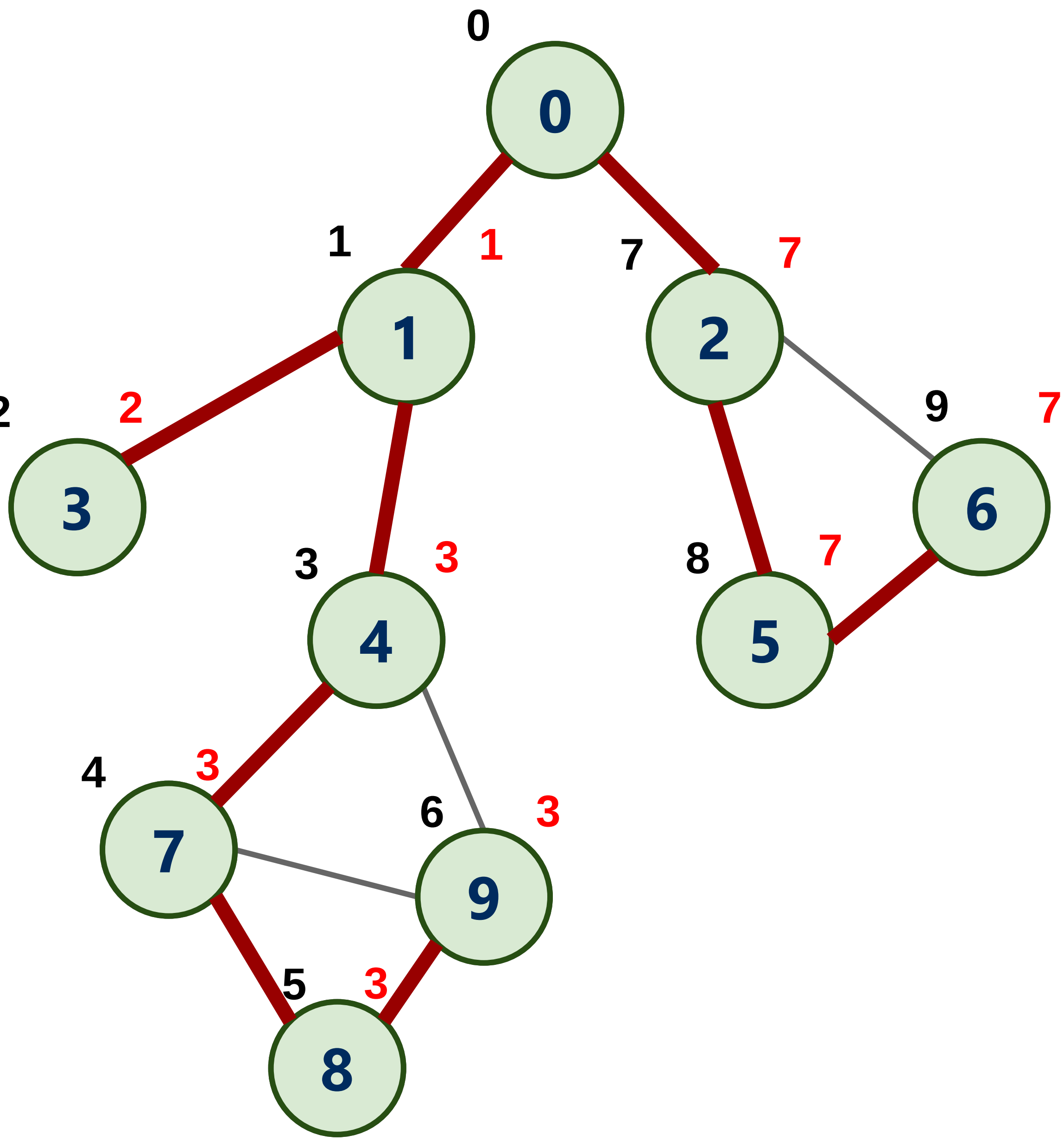


# low(v)

- low(v) = lowest-numbered vertex reachable from v using 0 or more spanning tree edges and then **at most one** back edge
  - Min of:
    - num(v) (the vertex is reachable from itself)
    - Lowest num(w) of all back edges (v, w)
    - Lowest low(w) of all children of *v* (the lowest-numbered vertex reachable through a child)

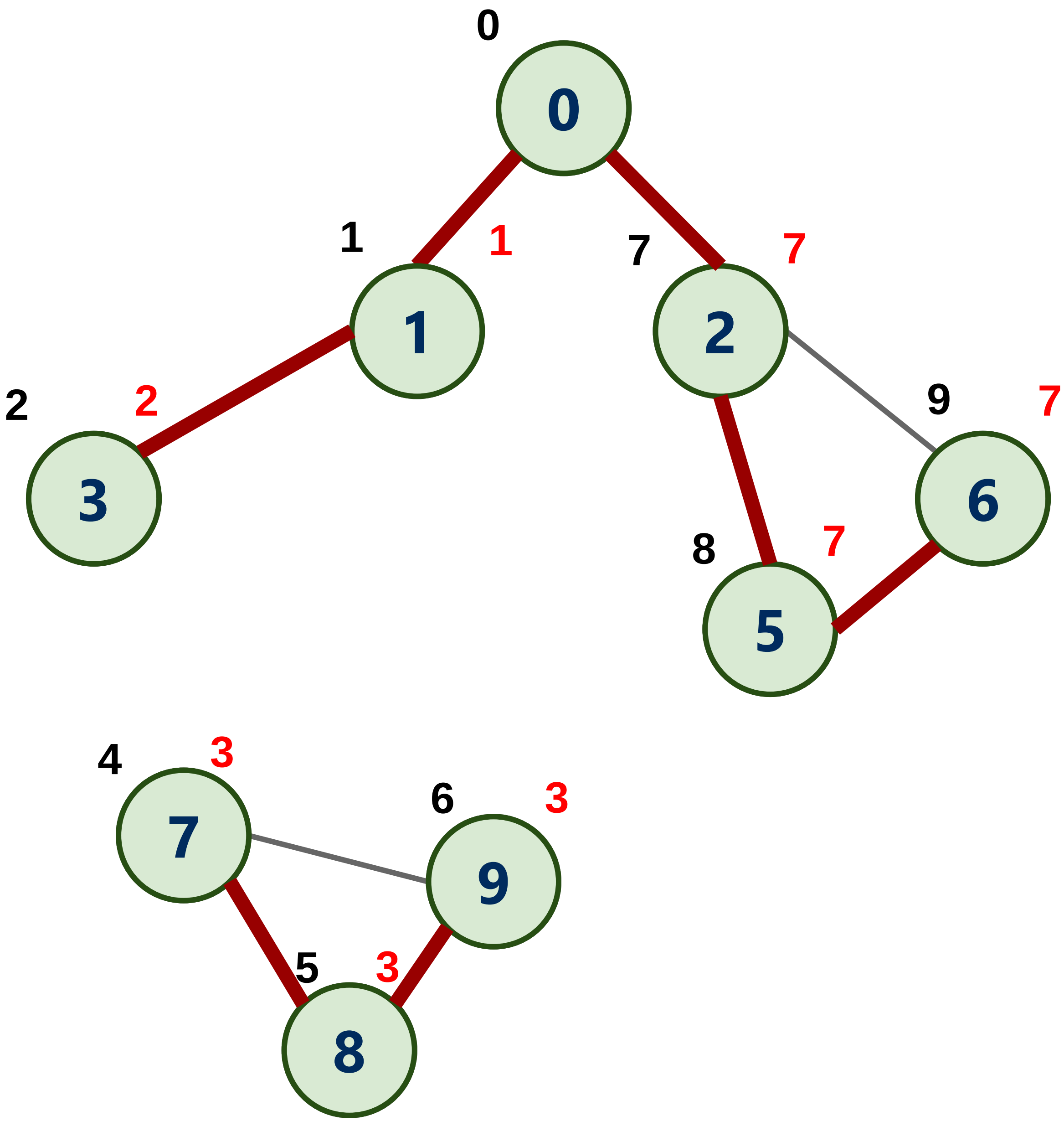
# Why are we computing low(v)?

- What does it mean if a vertex has a child such that
  - **low(child) >= num(parent)**?
- e.g., 4 and 7
- child has **no other way** except through parent to reach vertices with lower num values than parent
- e.g., 7 cannot reach 0, 1, and 3 except through 4
- So, the **parent is an articulation point**!
  - e.g., if 4 is removed, the graph becomes disconnected



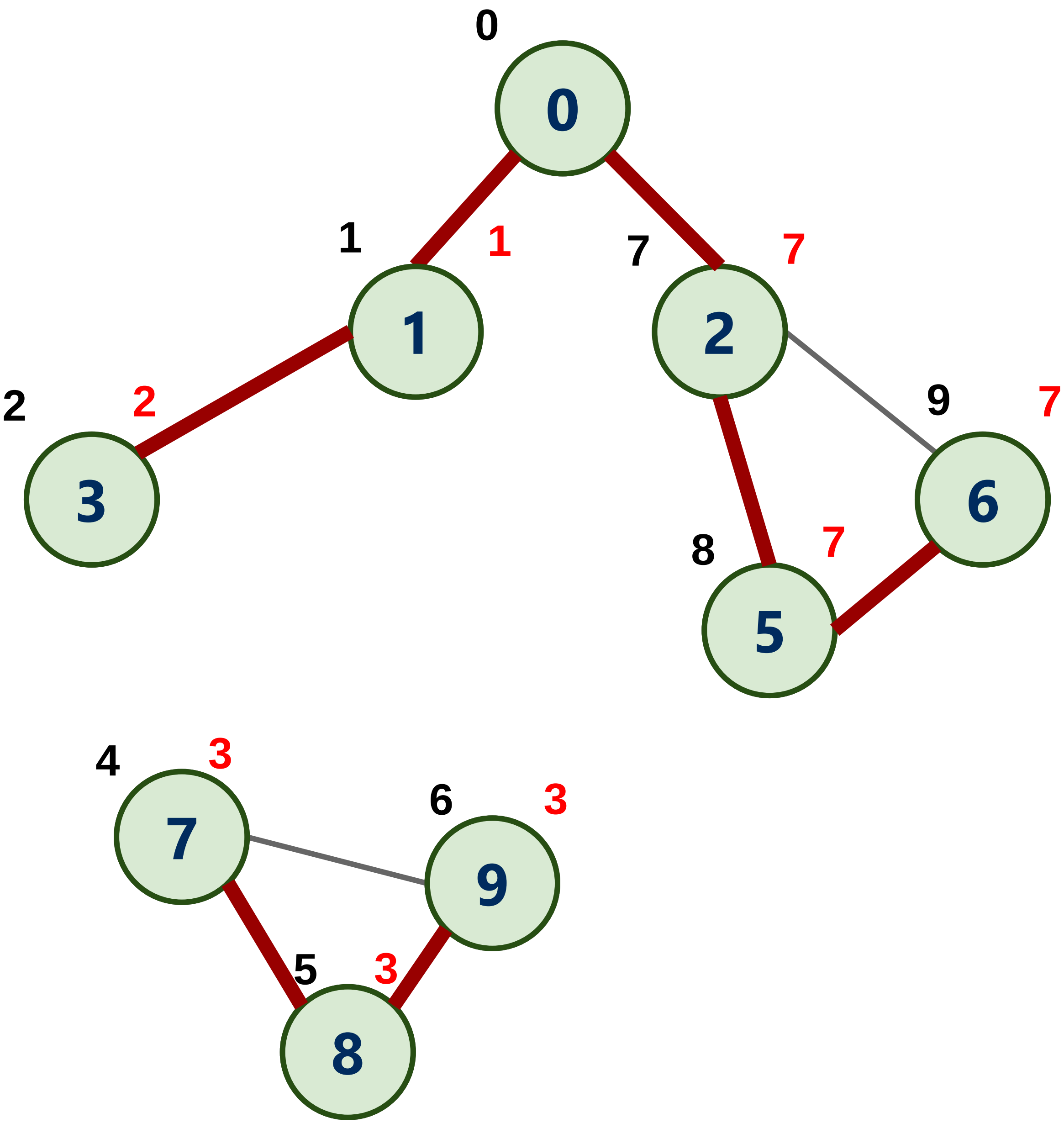
# Why are we computing low(v)?

- if 4 is removed, the graph becomes disconnected
- Each **non-root vertex v** that has a child w such that **low(w) >= num(v) is an articulation point**



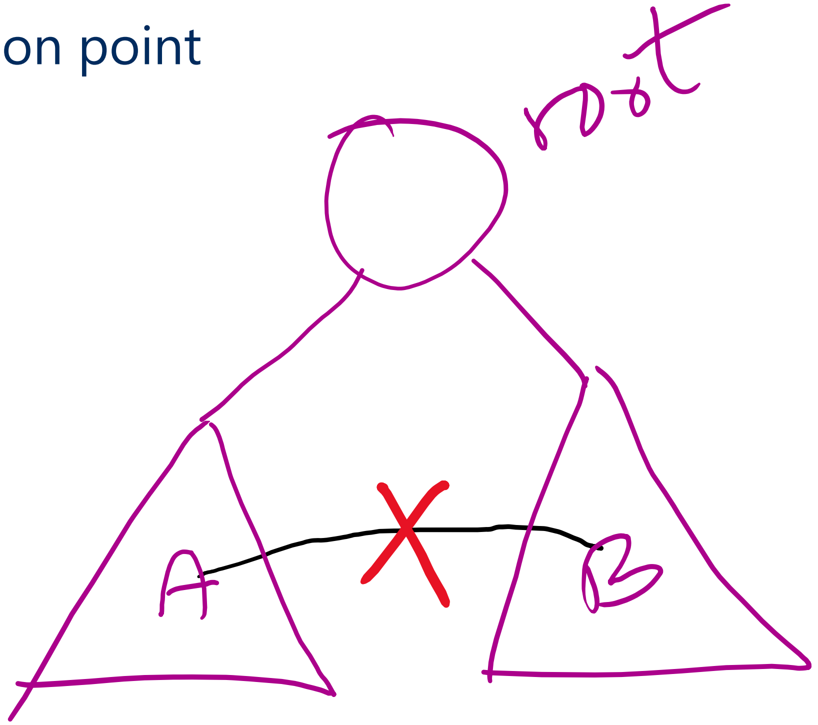
# What about the root vertex?

- The root has the smallest num value
  - root's children can't go "further" than root
- Possible that low(child) == num(root) but root is not an articulation point
- need a different condition for root



# What about the root of the spanning tree?

- What if we start DFS at an articulation point?
    - The starting vertex becomes the root of the spanning tree
    - If the root of the spanning tree has more than one child, the root is an articulation point



# Finding articulation points of a graph: The Algorithm

- As DFS visits each vertex v
  - Label v with with the two numbers:
    - num(v)
    - low(v): initial value is num(v)
  - For each neighbor w
    - if already seen → we have a back edge
      - update low(v) to num(w) if num(w) is less
    - if not seen → we have a child
      - call DFS on the child
      - **after the call returns,**
        - update low(v) to low(w) if low(w) is less

# when to compute num(v) and low(v)

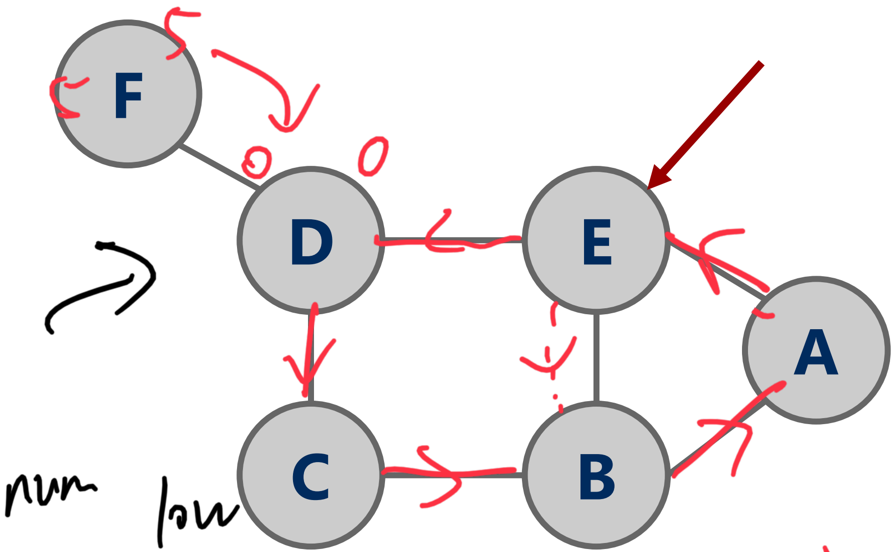
- num(v) is computed as we move down the tree
  - pre-order DFS
- low(v) is updated as we move down and up the tree
- Recursive DFS is convenient to compute both
  - why?

# Using DFS to find the articulation points of a connected undirected graph

```
int num = 0

DFS(vertex v) {
      num[v] = num++
      low[v] = num[v] //initially
      seen[v] = true //mark v as seen
      for each neighbor w
         if(w unseen){
            parent[w] = v
            DFS(w) //after the call returns low[w] is computed, why?
            low[v] = min(low[v], low[w])
            if(low[w] >= num[v]) v is an articulation point
         } else { //seen neighbor
            if(w != parent[v]) //and not the parent, so back edge
              low[v] = min(low[v], num[w])
         }
```

# Finding articulation points example




num low

0 D 0

1 C 0

2 B 0

3 A 0

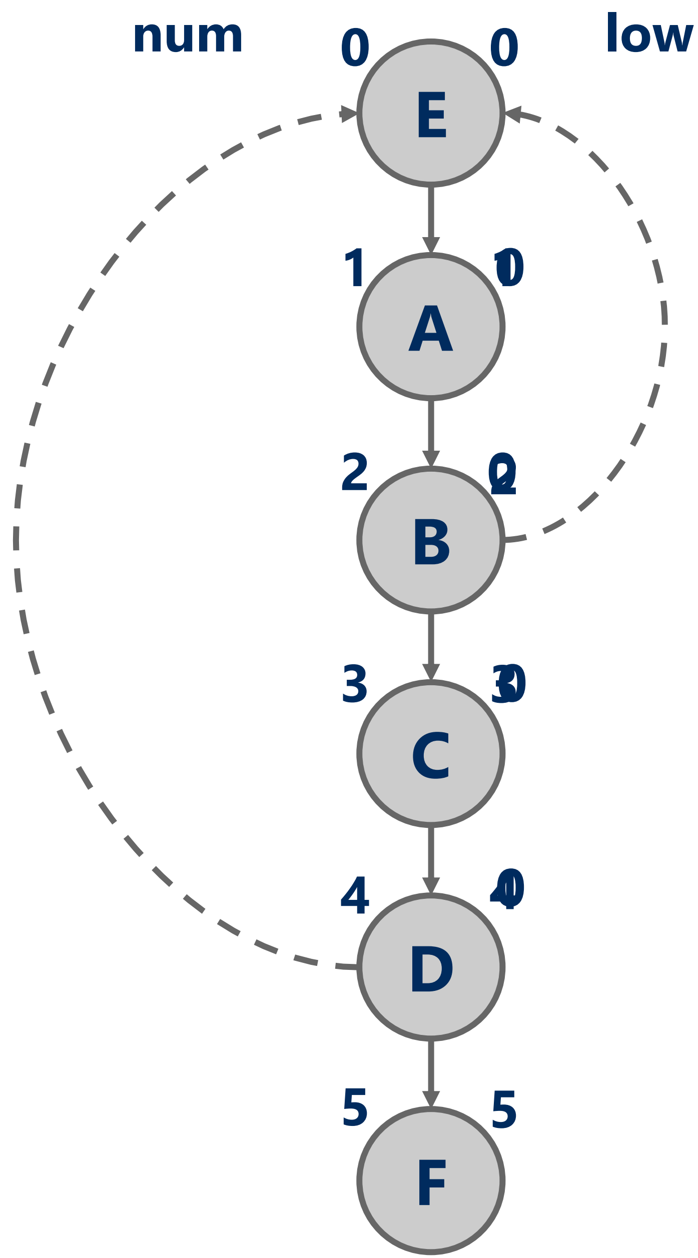
4 E 0

equal(or larger)

Articulation points

Last:

Check if root have more than 1 childs



# Repetitive Minimum Problem

- Input:
  - a (large) dynamic set of data items
- Output:
  - repeatedly find a minimum item
- You are implementing an algorithm that **repetitively** solve this problem
  - examples of such an algorithm?
    - Selection sort and Huffman tree construction
- What we cover today applies to the repetitive maximum problem as well

# Let’s create an ADT!

- The Priority Queue ADT

- Let’s generalize min and max to highest **priority**
- Primary operations of the PQ:
    - Insert
    - Find item with highest priority
        - e.g., findMin() or findMax()
    - Remove an item with highest priority
        - e.g., removeMin() or removeMax()
- We mentioned priority queues in building Huffman tries
- How do we implement these operations?
    - Simplest approach:  arrays

# Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - $\Theta(1)$
- Find:
  - Search for the highest priority item (e.g., min or max)
  - $\Theta(n)$
- Remove:
  - Search for the highest priority item and delete
  - $\Theta(n)$

# Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - $\Theta(n)$
- Find:
  - Return the item at the end of the array
  - $\Theta(1)$
- Remove:
  - Return and delete the item at the end of the array
  - $\Theta(1)$

## So what other options do we have?

- What about a balanced binary search tree?
  - Insert
    - $\Theta(\lg n)$
  - Find
    - $\Theta(\lg n)$
  - Remove
    - $\Theta(\lg n)$
- OK, all operations are $\Theta(\lg n)$
  - No constant time operations

# Which implementation should we choose?

- Depends on the application
- We can compare the *amortized runtime* of each implementation
- Given a set of operations performed by the application:

$$\text{Amortized runtime} = \frac{\text{Total runtime of a sequence of operations}}{\#\text{operations}}$$

# Example: Huffman Trie Construction

- K-1 iterations
  - K is the # unique characters in the file to be compressed
- Each iteration:
  - 2 removeMin calls
  - 1 insert call
- Unsorted Array: Total time Huffman Trie Construction =(K-1)*[2 * K + 1 * 1] = $O(K^2)$
- Sorted Array: Total time Huffman Trie Construction =(K-1)*[2 * 1 + 1 * K] = $O(K^2)$
- Balanced BST: Total time Huffman Trie Construction =(K-1)*[2 * log K + 1 * log K] = O(K log K)

# Repetitive Highest Priority Problem

- Input:
  - a (large) dynamic set of data items
    - each item has a priority
    - e.g., highest priority is minimum item
    - e.g., highest priority is maximum item
  - a *stream* of zero or more of each of the following operations
    - Find a highest priority item in the set
    - Insert an item to the set
    - Remove a highest priority item from the set
- Examples
  - Selection sort
    - Repeatedly, remove a minimum item from the array and insert it in its correct position in the sorted array
  - Huffman trie construction
    - Each iteration: remove a minimum tree from the forest (**twice**) and insert a new tree

# Let’s create an ADT!

- The ADT Priority Queue (PQ)

- Primary operations of the PQ:
  - Insert
  - Find item with highest priority
    - e.g., findMin() or findMax()
  - Remove an item with highest priority
    - e.g., removeMin() or removeMax()

# What are possible implementations of the PQ ADT?

| | findMin | removeMin | insert |
|---|---|---|---|
| Unsorted Array | O(n) | O(n) | O(1) |
| Sorted Array | O(1) | O(1) | O(n) |
| Red-Black BST | O(log n) | O(log n) | O(log n) |

# Is a BST overkill to implement ADT PQ?

- Balanced BST (e.g., RB-BST) provides *log n* runntime time for all operations
- Our find and remove operations only need the highest priority item, not to find/remove *any* item
    - Can we take advantage of this to improve our runtime?
        - Yes!

# The heap

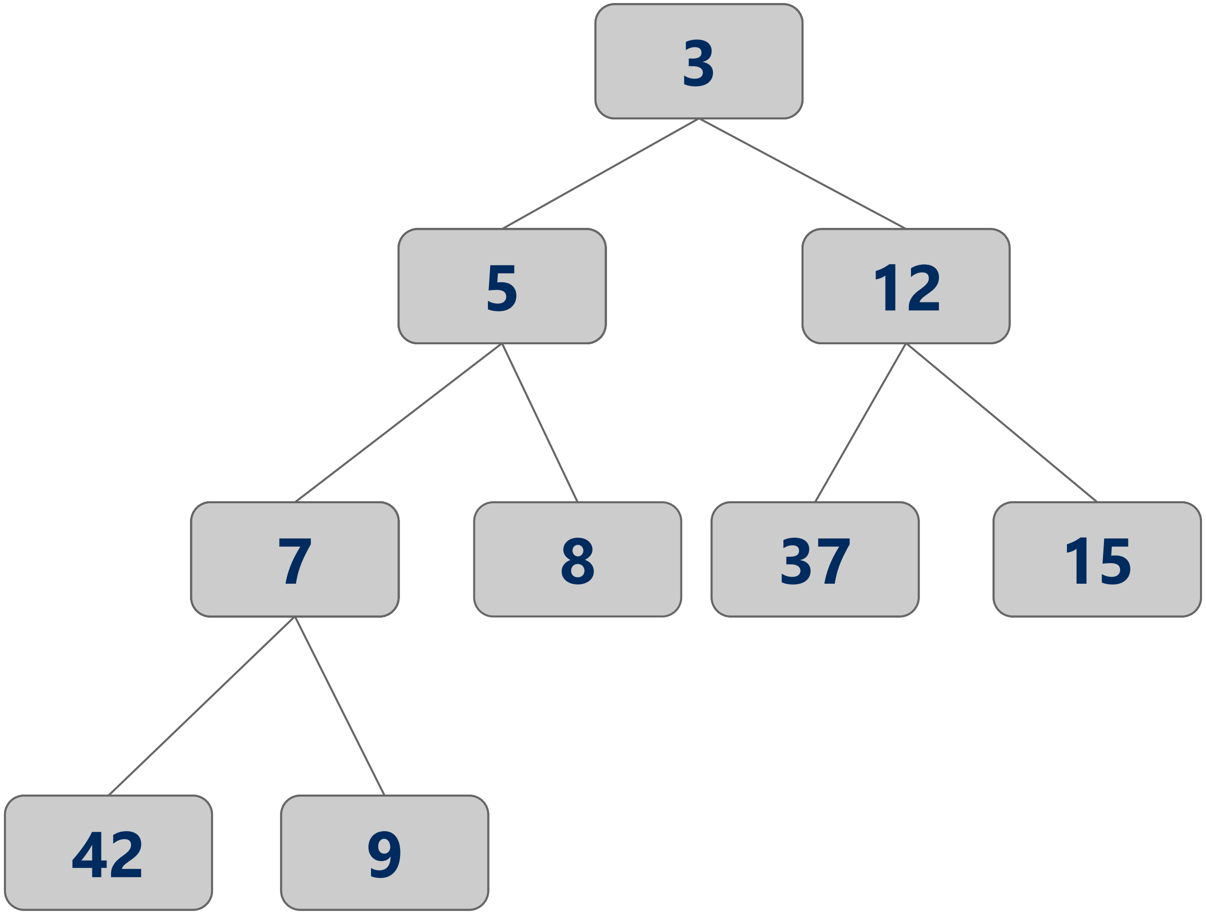
- A heap is **complete** binary tree such that for each node T in the tree:
  - T.item is of a higher priority than T.right_child.item
  - T.item is of a higher priority than T.left_child.item

- It does not matter how T.left_child.item relates to T.right_child.item
  - This is a relaxation of the approach needed by a BST

The *heap property*

# Min Heap Example

- In a Min Heap, a highest priority item is a minimum item



# Heap PQ runtimes

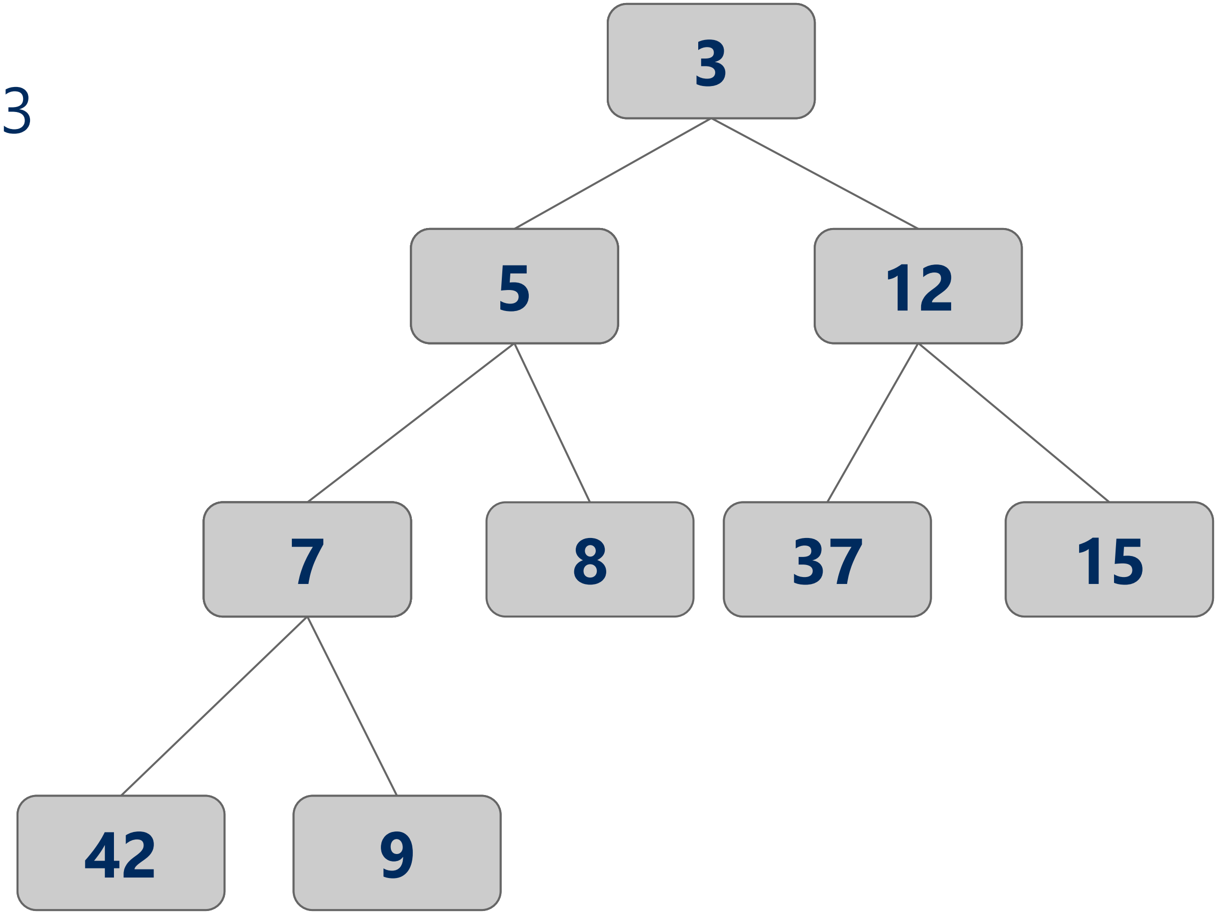
- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

# Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

# Min heap insert

Insert:
7, 42, 37, 5, 8, 15, 12, 9, 3

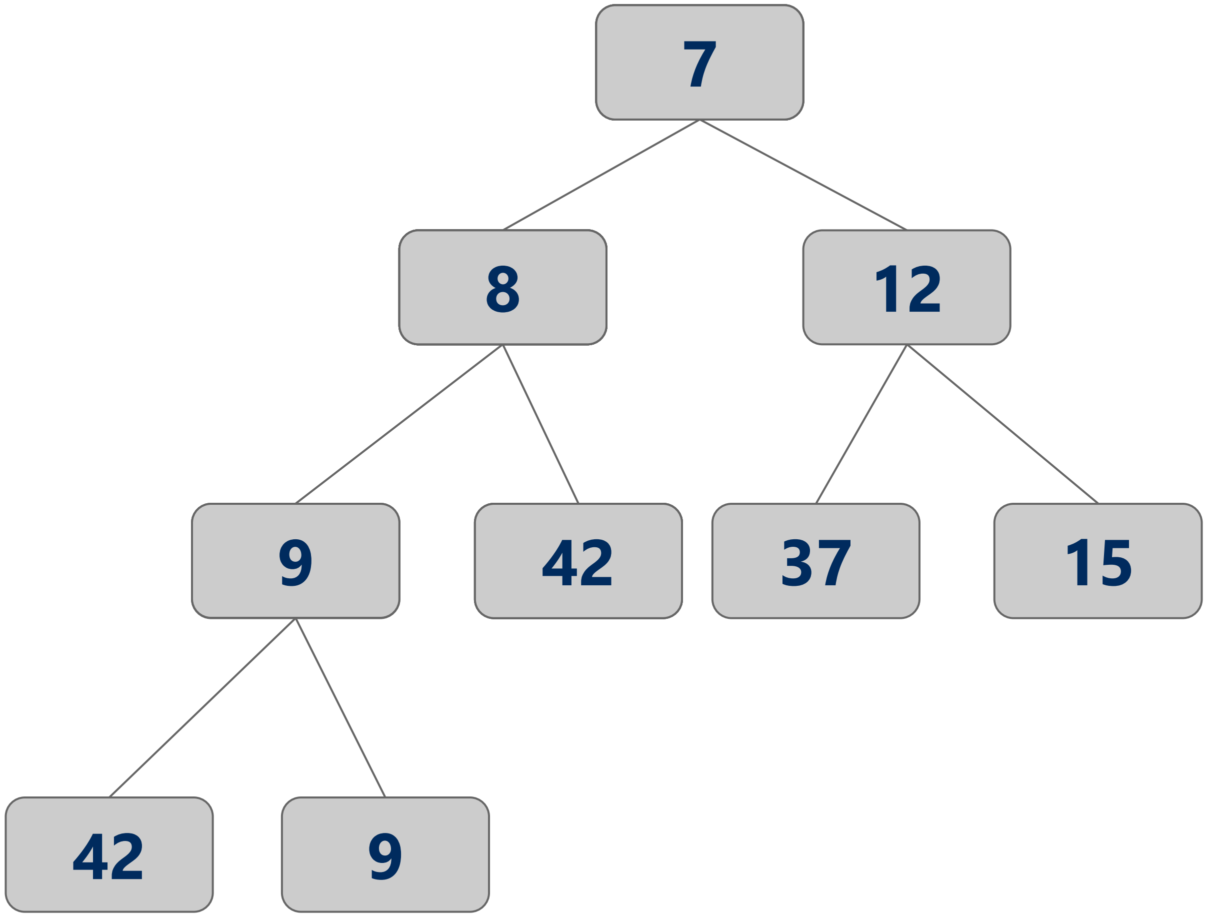


# Heap remove

- Tricky to delete root...
  - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
    - But then the root is violating the heap property...
      - So we push the root down the tree until it is supporting the heap property

# Min heap removal

**NO!**



# Heap runtimes

- Find
  - $\Theta(1)$
- Insert and remove
  - Height of a complete binary tree is lg n
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are $\Theta(\lg n)$

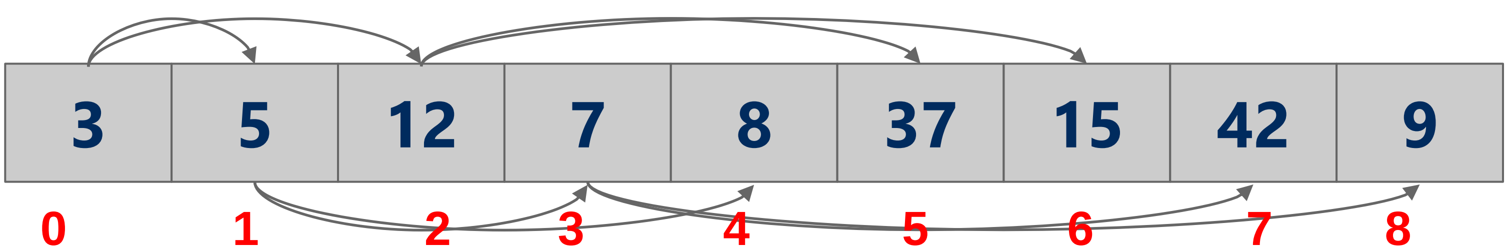
# Heap implementation

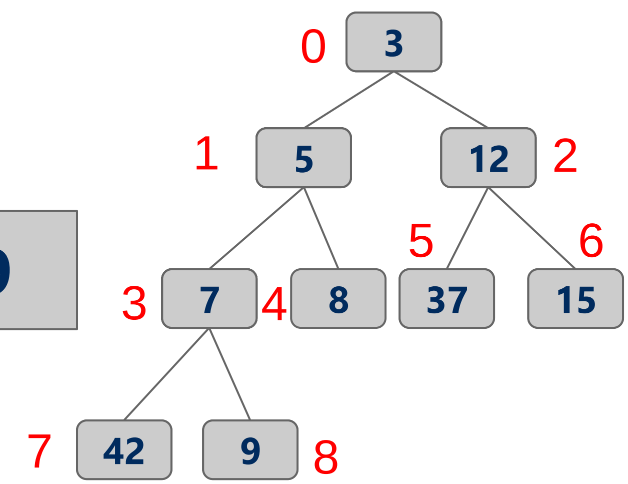
- Simply implement tree nodes like for BST
    - This requires overhead for dynamic node allocation
    - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree…
    - We can easily represent a complete  binary tree using an array

# Storing a heap in an array

- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index i
  - parent(i) = $\lfloor(i - 1) / 2\rfloor$
  - left_child(i) = 2i + 1
  - right_child(i) = 2i + 2

For arrays indexed from 0

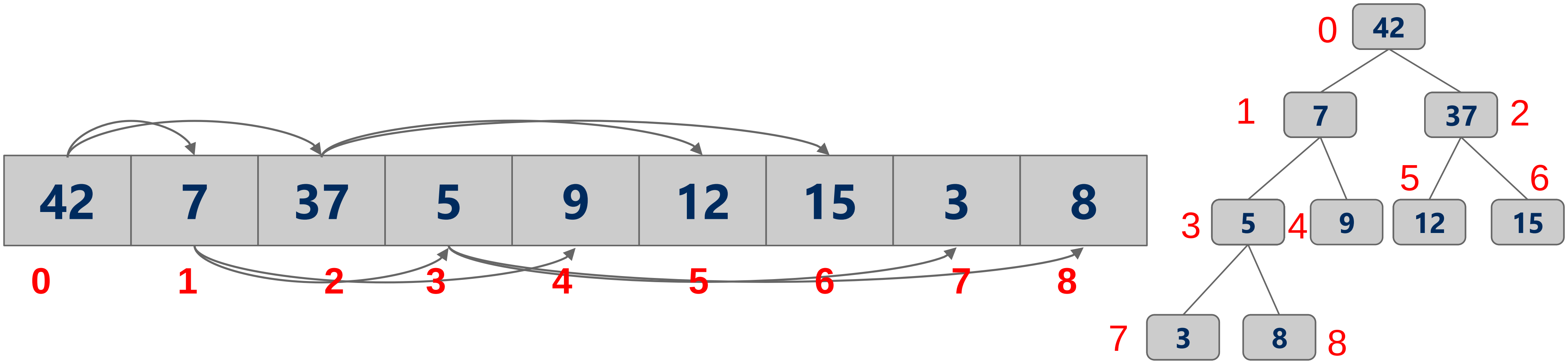


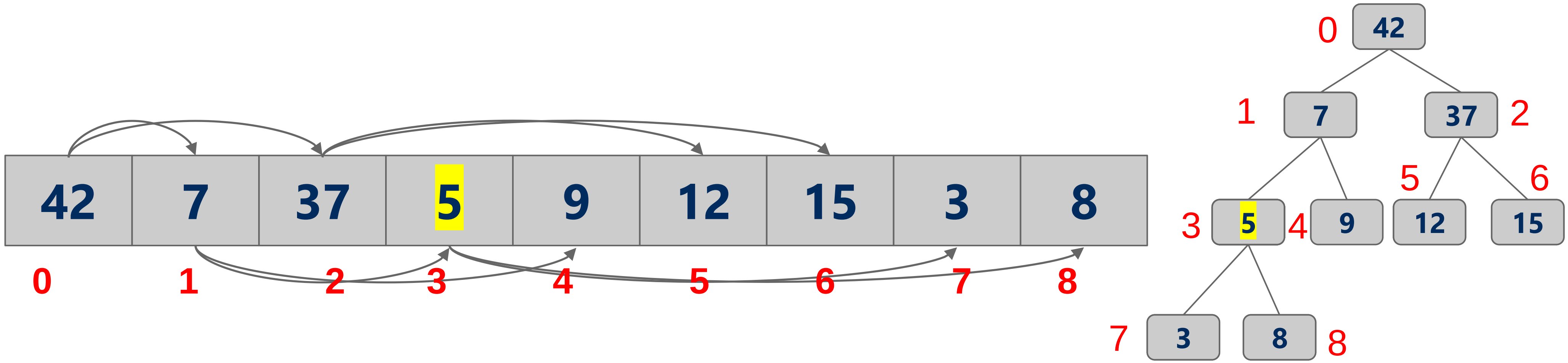
# Can we turn any array into a heap?

- Yes!
- Any array can be thought of as a complete tree!
- We can change it into a heap using the following algorithm
- Scan through the array **right to** left starting from the rightmost non-leaf
  - the largest index *i* such that left_child(i) is a valid index (i.e., < n)
  - $2i+1 < n \rightarrow i < (n-1)/2$
  - push the node down the tree until it is supporting the heap property
- This is called the **Heapify** operation

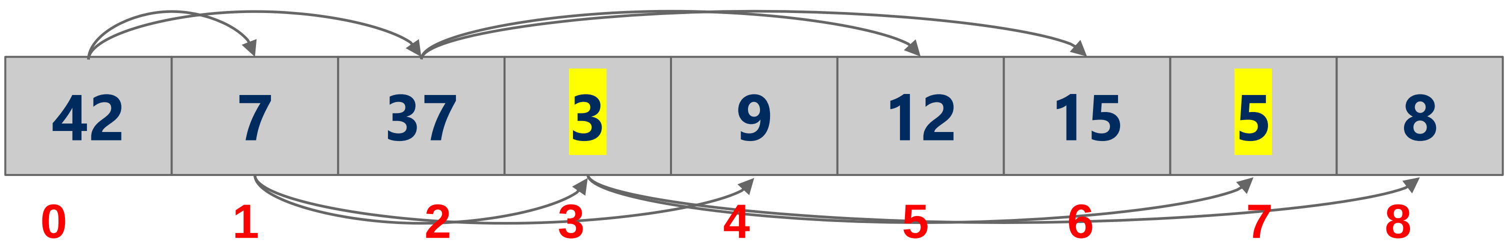
# Heapify Example: Building a Min Heap

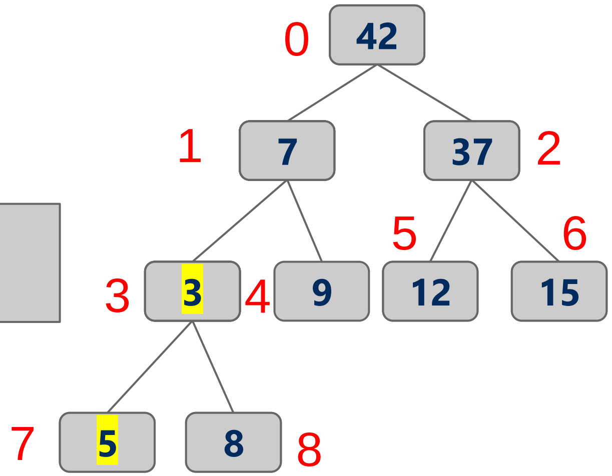


# Heapify Example: Building a Min Heap

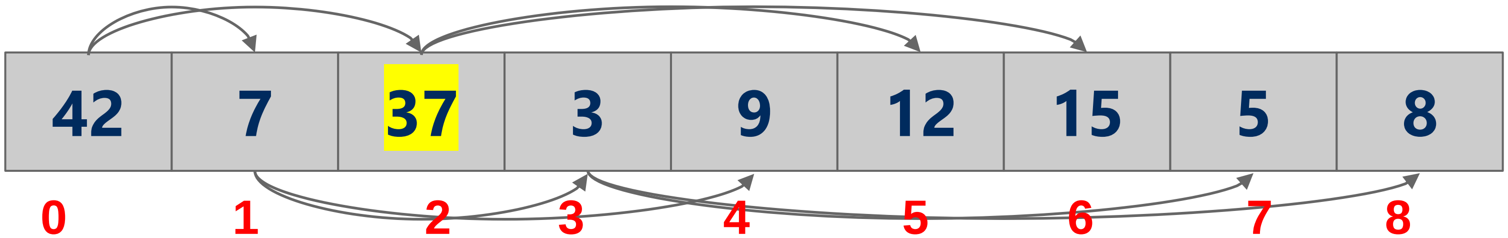


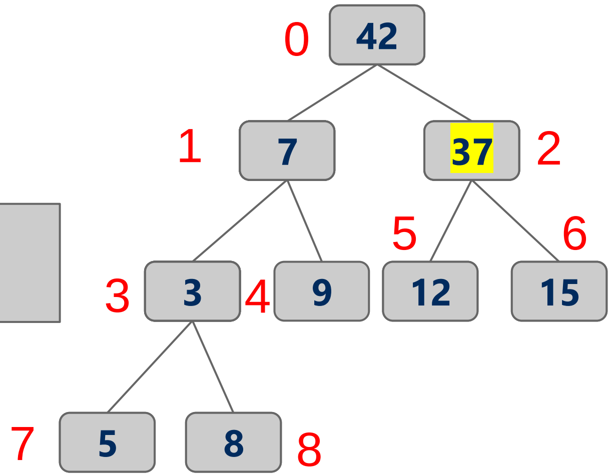
# Heapify Example: Building a Min Heap



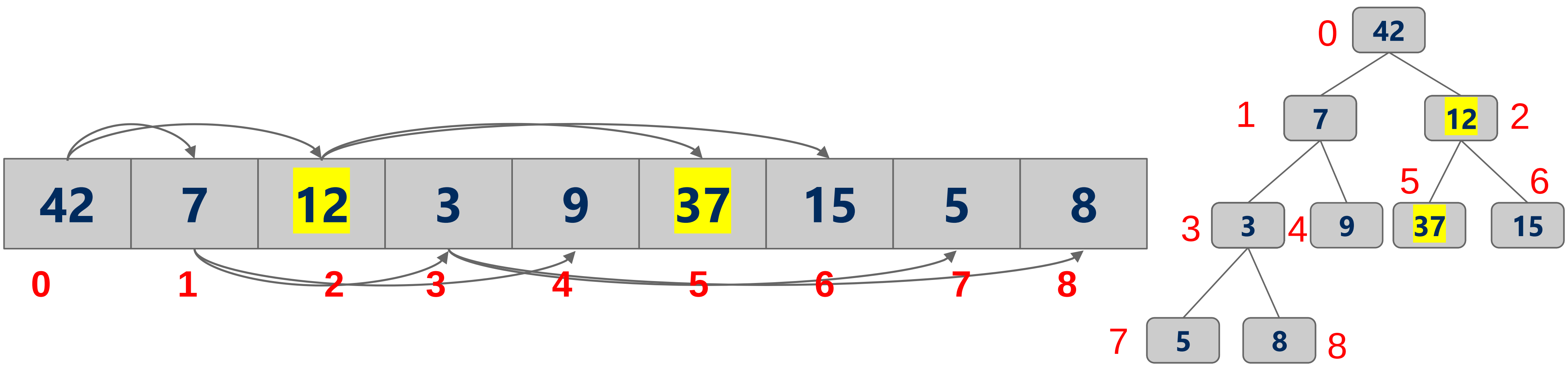



# Heapify Example: Building a Min Heap

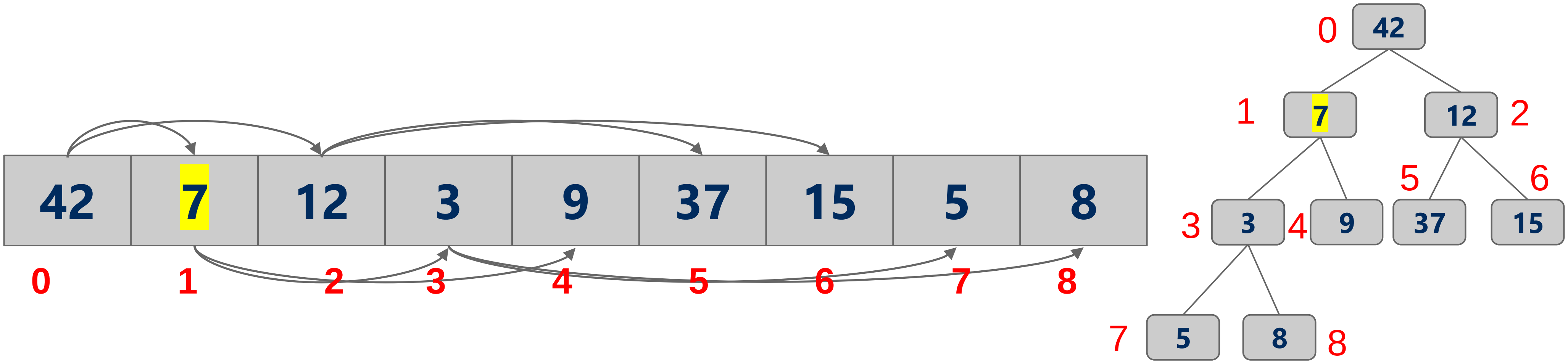


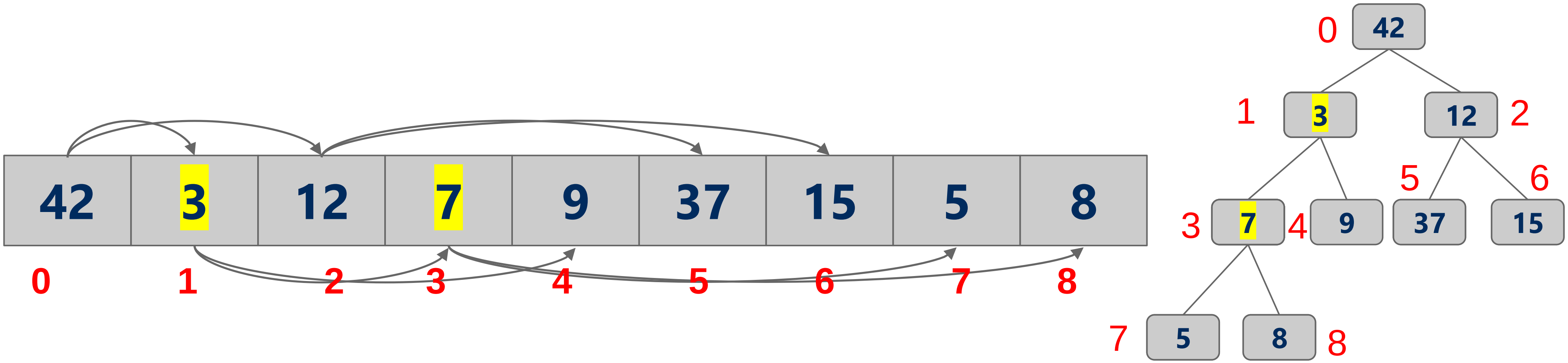
# Heapify Example: Building a Min Heap



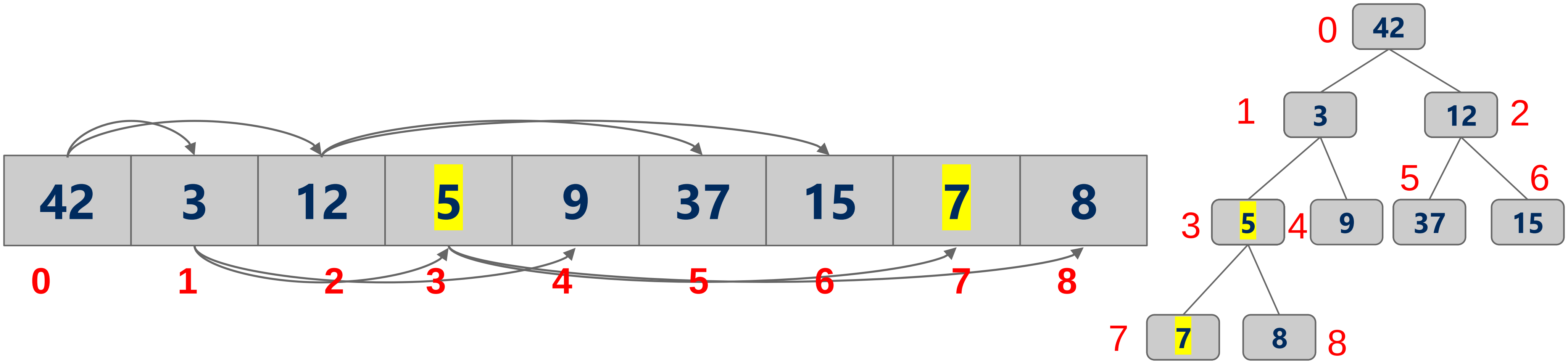
# Heapify Example: Building a Min Heap



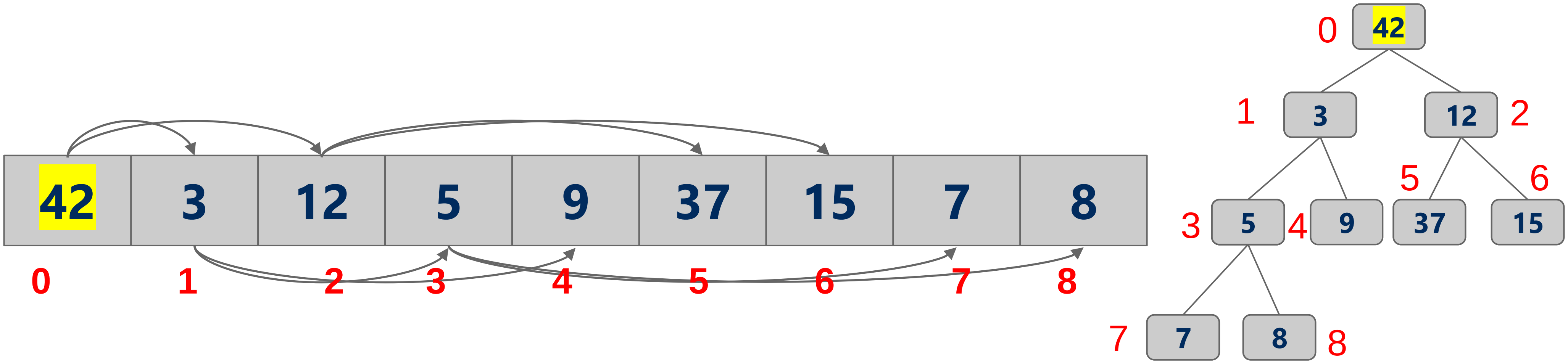
# Heapify Example: Building a Min Heap



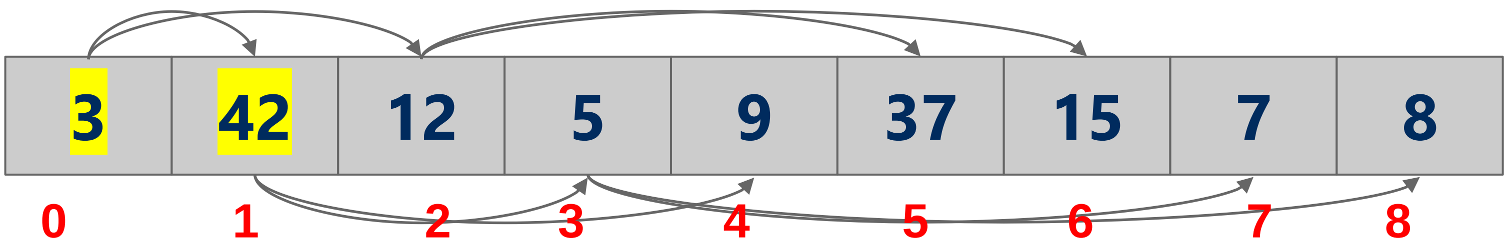
# Heapify Example: Building a Min Heap

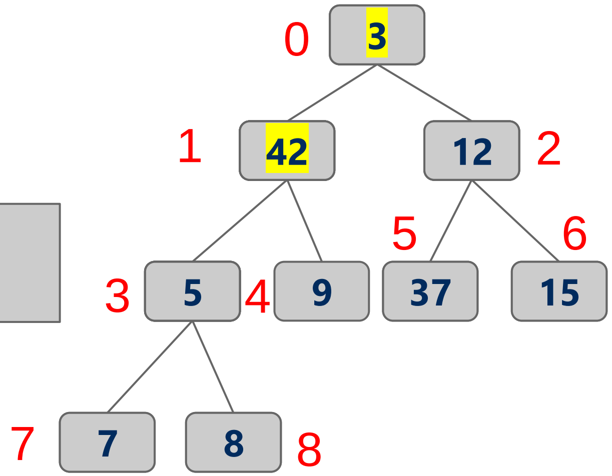


# Heapify Example: Building a Min Heap

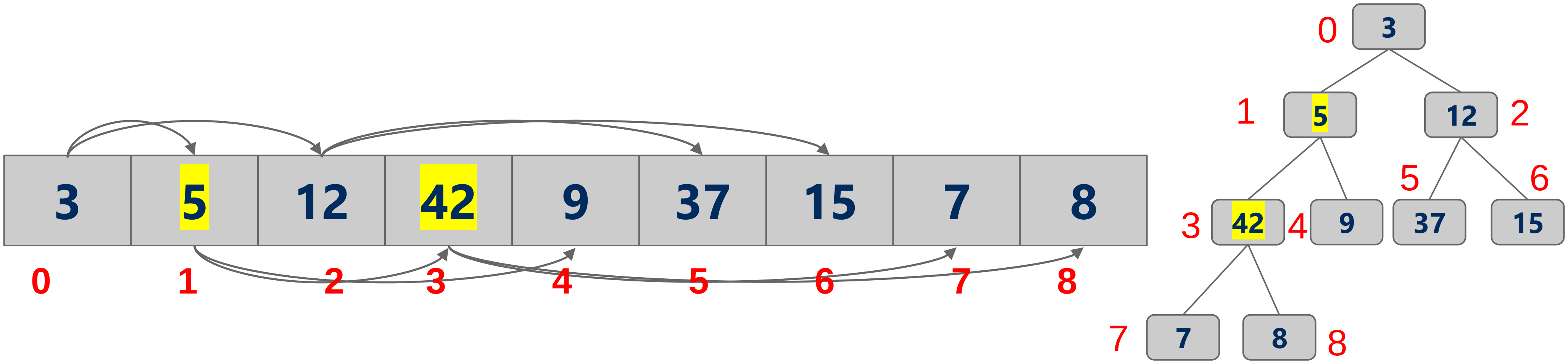


# Heapify Example: Building a Min Heap

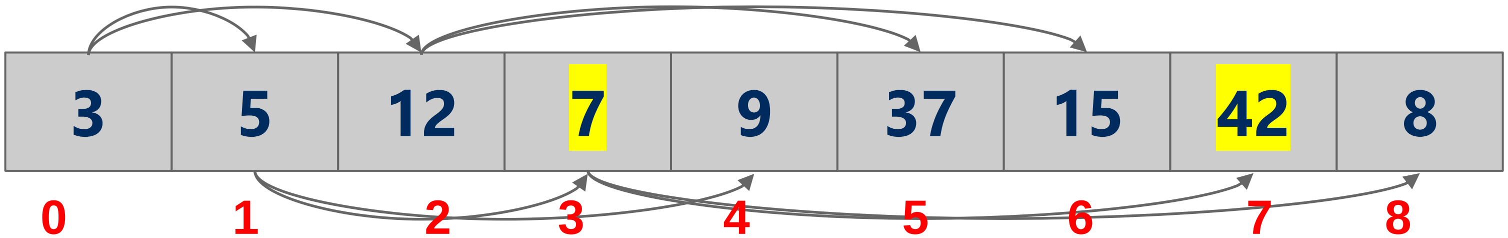


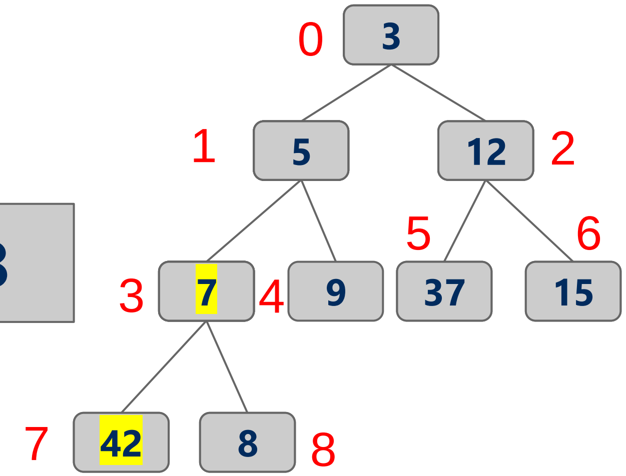



# Heapify Example: Building a Min Heap

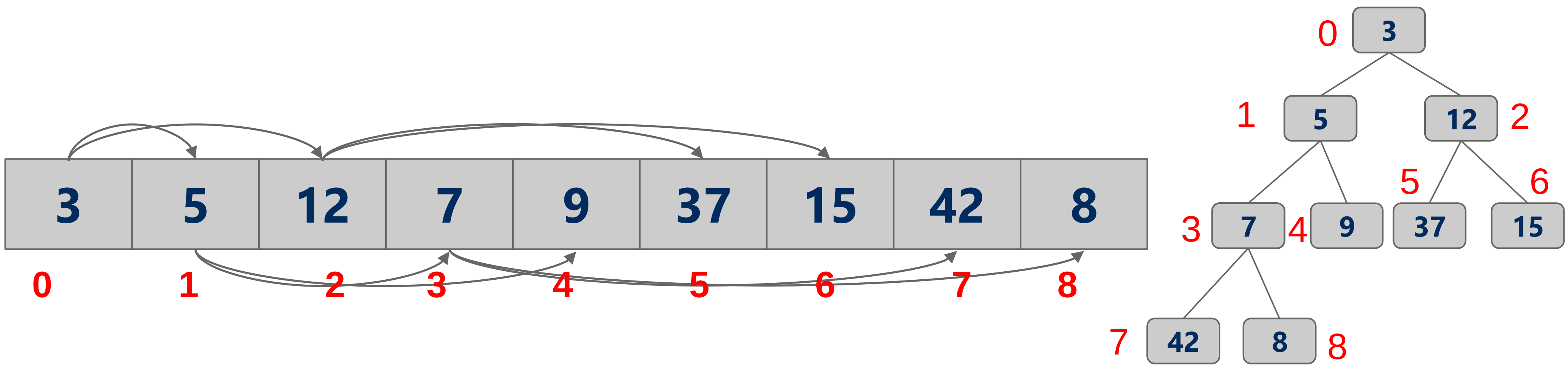


# Heapify Example: Building a Min Heap



# Heapify Example: Building a Min Heap

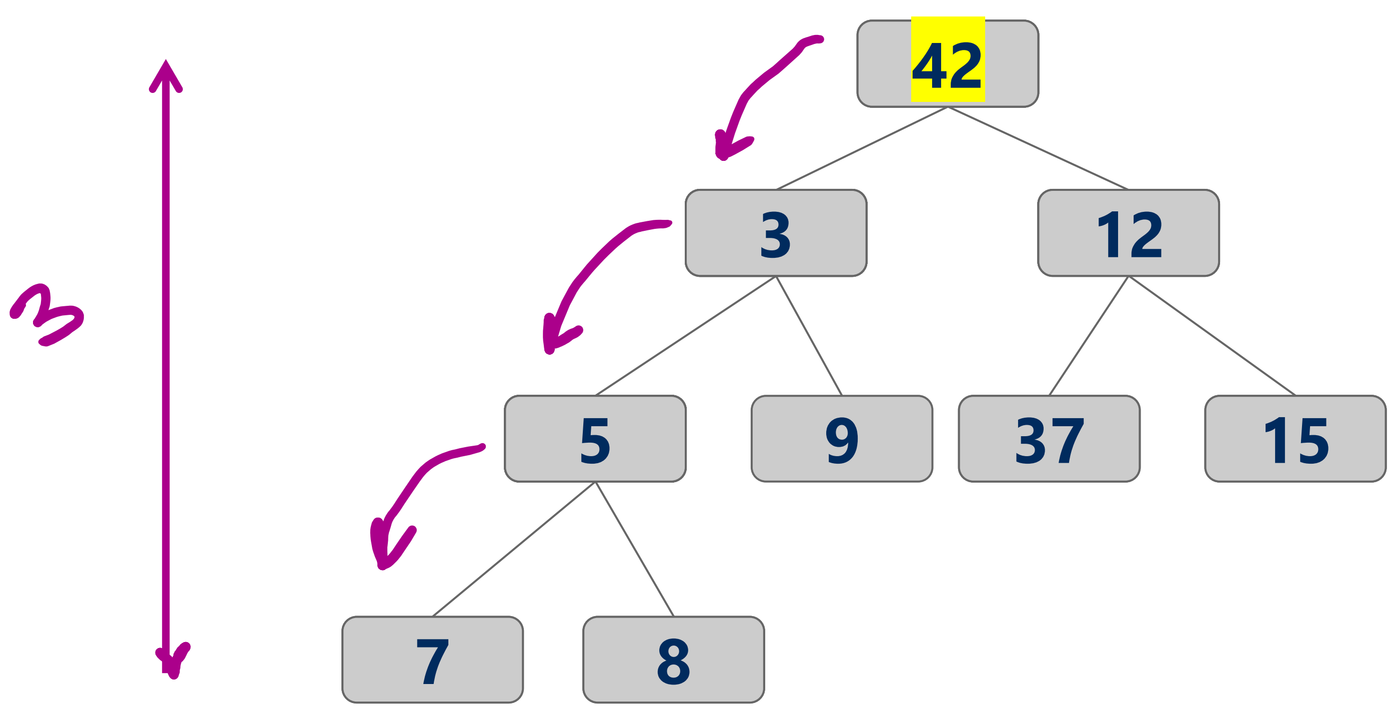


# Heapify Running time

- Upper bound analysis:
    - We make about n/2 downheap operations
        - log n each
    - So, O(n log n)

# Heapify Running time

- A tighter analysis
    - for each node that we start from, we make at most *height[node]* swaps



# Heapify Running time: A tighter analysis

- $Runtime = \sum_{i=1}^{n} height[n]$
- $= \sum_{i=0}^{\log n} number\ of\ nodes\ with\ height\ i$
- Assume a full tree
  - A node with height *i* has $2^i$ nodes in its subtree including itself
  - Assume *k* nodes with height *i*:
  - they will have $k2^i$ nodes in their subtrees
  - $k2^i <= n \rightarrow k <= n/2^i$
- So, at most $n/2^i$ nodes exist with height *I*
- $\sum_{i=o}^{\log n} \frac{n}{2^i} = n + \frac{n}{2} + \frac{n}{4} + \ \ldots$
- $= \ \theta(largest\ term) = \ \theta(n)$

# Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don’t actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat

# Heap sort analysis

- Runtime:
  - Worst case:
    - n log n
- In-place?
  - Yes
- Stable?
  - No

# Storing Objects in PQ

- What if we want to **<u>update</u>** an Object in the heap?
  - What is the runtime to find an arbitrary item in a heap?
    - Θ(n)
    - Hence, updating an item in the heap is Θ(n)
  - Can we improve of this?
    - Back the PQ with something other than a heap?
    - Develop a clever workaround?

# Indirection

- Maintain a second data structure that maps item IDs to each item's current position in the heap
- This creates an *indexable* PQ

# Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

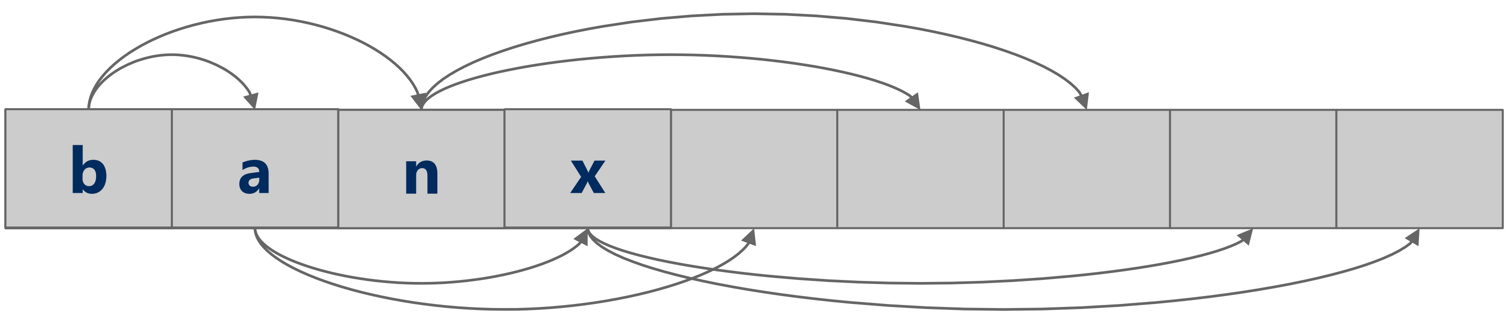
```
class CardPrice implements Comparable<CardPrice>{
      public String store;
      public double price;
      public CardPrice(String s, double p) { … }
      public int compareTo(CardPrice o) {
            if (price < o.price) { return -1; }
            else if (price > o.price) { return 1; }
            else { return 0; }
      }
}
```

# Indirection example

- n = new CardPrice("NE", 333.98);
- a = new CardPrice("AMZN", 339.99);
- x = new CardPrice("NCIX", 338.00);
- b = new CardPrice("BB", 349.99);

- Update price for NE:  340.00
- Update price for NCIX:  345.00
- Update price for BB:  200.00

| Indirection |
| --- |
| "NE":2 |
| "AMZN":1 |
| "NCIX":3 |
| "BB":0 |



# Indexable PQ Discussion

- How are our runtimes affected?
- space utilization?
- how should we implement the indirection?
- what are the tradeoffs?

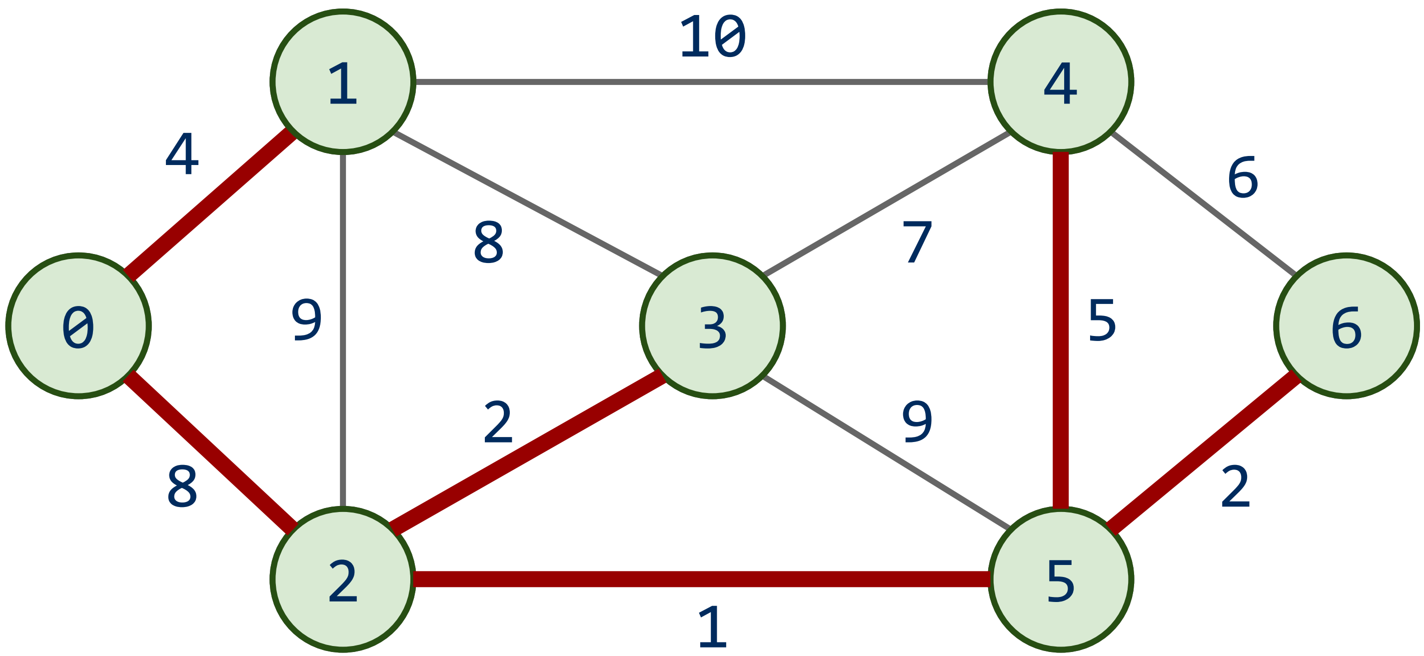
# Neighborhood connectivity Problem

- keep a set of neighborhoods connected
  - We can go from any neighborhood to any other
- with the minimum cost possible
- **Input:** A set of neighborhoods and a file with the following format:
  - neighborhood i, neighborhood j, cost of connecting the two neighborhoods
  - …
- **Output:** A set of neighborhood pairs to be connected and a total cost such that
  - Neighborhoods are connected
  - The total cost is minimum

# Prim’s algorithm

- Initialize T  to contain the starting vertex
  - T will eventually become the MST
- While there are vertices not in T:
  - Find minimum edge-weight edge that connects a vertex in T to a vertex not yet in T
  - Add the edge with its vertex to T

# Prim's algorithm



# Runtime of Prim's

- At each step, check all possible edges
- For a complete graph:
  - First iteration:
    - v - 1 possible edges
  - Next iteration:
    - 2(v - 2) possibilities
      - Each vertex in T shared v-1 edges with other vertices, but the edges they shared with each other already in T
  - Next:
    - 3(v - 3) possibilities
  - ...
- Runtime:
  - $\Sigma_{i\ =\ 1\ \text{to}\ v-1}$ (i * (v - i)) = Θ(largest term * number of terms)
  - number of terms = *v-1*
  - largest term is $v^2/4$ (when *i=v/2)*
  - Evaluates to $\Theta(v^3)$

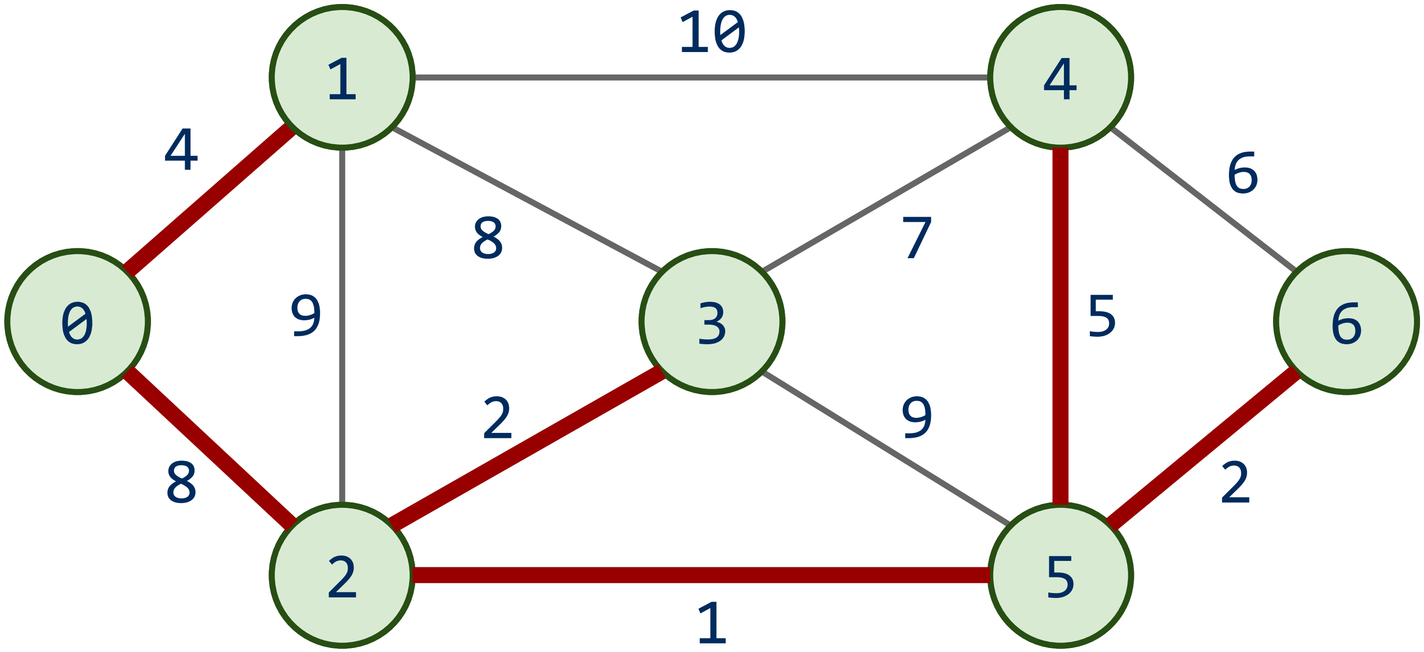
# Do we need to look through all remaining edges?

- No!  We only need to consider the *best* edge possible for each vertex!
  - The best edge of each vertex can be updated as we add each vertex to T

# An enhanced implementation of Prim's Algorithm

- Add start vertex to T
- Search through the neighbors of the added vertex to adjust the parent and best edge arrays as needed
- Search through the best edge array to find the next addition to T
- Repeat until all vertices added to T

# Prim's algorithm




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| Parent: | -- | 0 | 0 | 2 | 5 | 2 | 5 |
| Best Edge: | 0 | 4 | 8 | 2 | 5 | 1 | 2 |